Title

Applying an abstract data structure description approach to parallelizing scientific pointer programs

Authors

Hummel, Joseph
Nicolau, Alexandru
Hendren, Laurie J.

Publication Date

1992-01-27

Peer reviewed

Applying an Abstract Data Structure Description Approach to Parallelizing Scientific Pointer Programs

TECHNICAL REPORT 92-10

Joseph Hummel*
Alexandru Nicolau†, and
Laurie J. Hendren‡

January 27, 1992

*UC-Irvine, Dept. of ICS, Irvine, CA 92717-3425.

†UC-Irvine, Dept. of ICS. This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

‡McGill University, School of Computer Science. This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

Applying an Abstract Data Structure Description Approach to Parallelizing Scientific Pointer Programs

Joseph Hummel
Alexandru Nicolau *
Dept. of Information and Computer Science
UC-Irvine

Laurie J. Hendren †
School of Computer Science
McGill University

Correspondence to:
Joseph Hummel
Dept. of ICS
UC-Irvine
Irvine, CA 92717
Phone: (714) 725-2248
E-mail: jhummel@ics.uci.edu

Keywords:
Compiler Parallelization, Data Structure Abstraction, Data Structure Analysis, Optimizing Transformations, Scientific Pointer Programs.

Abstract

Even though impressive progress has been made in the area of parallelizing scientific programs with arrays, the application of similar techniques to programs with pointer data structures has remained difficult. Unlike arrays which have a small number of well-defined properties that can be utilized by a parallelizing compiler, pointer data structures are used to implement a wide variety of structures that exhibit a much more diverse set of properties. The complexity and diversity of such properties means that, in general, scientific programs with pointer data structures cannot be effectively analyzed by an optimizing and parallelizing compiler.

In order to provide a system in which the compiler can fully utilize the properties of different types of pointer data structures, we have developed a mechanism for the Abstract Description of Data Structures (ADDS). With our approach, the programmer can explicitly describe important properties such as dimensionality of the pointer data structure, independence of dimensions, and direction of traversal. These abstract descriptions of pointer data structures are then used by the compiler to guide analysis, optimization, and parallelization.

In this paper we summarize the ADDS approach through the use of numerous examples of data structures used in scientific computations, we illustrate how such declarations are natural and non-tedious to specify, and we show how the ADDS declarations can be used to improve compile-time analysis. In order to demonstrate the viability of our approach, we show how such techniques can be used to parallelize an important class of scientific codes which naturally use recursive pointer data structures. In particular, we use our approach to develop the parallelization of an N-body simulation that is based on a relatively complicated pointer data structure, and we report the speedup results for a Sequent multiprocessor.

*This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.
†This work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

1 Introduction and Motivation

Scientific codes have often been the target of optimizing and parallelizing compilers. Such codes typically use arrays for storing data, and loops with regular indexing properties to manipulate these arrays. A good deal of work has been done in the area of analysis and transformation in the presence of arrays and loops and as a result numerous optimizing and parallelizing techniques, such as invariant code motion, induction variable elimination, loop unrolling, vectorizing, prefetching, and various instruction scheduling strategies such as software pipelining and delay slot filling, have been developed [Kuc78, DH79, RG82, PW86, AK87, ASU87, AN88, Lam88, ZC90].

Unfortunately, there has not been as much progress in the area of parallelizing programs that use pointer data structures. Unlike arrays which have a small number of well-defined properties that can be utilized by a parallelizing compiler, pointer data structures are used to implement a wide variety of structures that exhibit a much more diverse set of properties. For example, structures may be linear (lists), hierarchical (trees), or even cyclic (graphs). In addition, programs using pointer data structures often use recursion rather than looping constructs. These complexities mean that programs which utilize dynamically-allocated pointer data structures are much more difficult to analyze, hindering progress in this area.

This lack of progress is problematic, since numerous data structures in scientific programs—sparse matrices [Sta80] and quadtrees [Sam90] for example—are typically built using recursively-defined pointer data structures. Pointer data structures are useful in a variety of scientific codes including computational geometry [Sam90] and the so-called *tree-codes* [App85, BH86]. Furthermore, with the increased use of languages supporting pointers (such as C and Fortran 90) for scientific computing, it is expected that the number of programs using pointer data structures will increase.

In order to understand and attack these problems, the goal of this paper is to: (1) outline the importance and difficulty of accurate alias analysis for general pointer data structures; (2) summarize our approach to this problem, that is the development of abstract descriptions of pointer data structures; (3) show how such abstract descriptions can aid in alias analysis and parallelizing transformations; and (4) illustrate the approach by parallelizing a typical tree-code based scientific program.

2 The Challenge of Alias Analysis

The combination of complex pointer data structures with diverse properties and complex recursive control flow has provided new challenges for parallelizing compilers. One of the key challenges is *alias analysis*. This analysis is used to detect when two distinct memory accesses may refer to the same physical memory location. Alias analysis is a critical part of parallelizing compilers, and the effectiveness of many compiler analysis techniques and parallelizing transformations rely upon accurate alias analysis.

2.1 Previous Approaches

In dealing with complex pointer data structures, there have been two approaches for dealing with alias analysis: (1) concentrate on analyzing arrays, and make overly conservative assumptions for all pointer data structures, and (2) develop static structure analysis techniques for a class of pointer data structures. Although approach (1) is far more common, there has been significant work on approach (2). One class of solutions has been the development of advanced alias analysis techniques (also called structure estimation techniques) that attempt to statically approximate dynamically-allocated data structures with some abstraction. The most commonly used abstraction has been *k-limited* graphs [JM81], and variations on *k-limited graphs* [LH88a, LH88b, HPR89, CWZ90]. The major disadvantage of these techniques is that the approximation introduces cycles in the abstraction, and thus making it difficult to distinguish list or tree-like data structures from data structures that truly contain cycles. This is a serious disadvantage for parallelization based on these approximations since it is precisely this acyclic nature of list and tree-like data structures that allows the application of many parallelizing transformations. The work by Chase et al. [CWZ90] has addressed this problem to some degree; however, their method fails to find accurate structure estimates in the presence of general recursion. This is a serious drawback since many programs heavily utilize recursion. Another method, *path matrix* analysis, was designed to specifically deal with distinguishing tree-like data structures from DAG-like (shared) and graph-like (cyclic) structures [HN90, Hen90]. This analysis uses the special properties exhibited by tree-like structures to provide a more accurate analysis of list and tree-like structures even in the presence of recursion. However, it has the disadvantage that it cannot handle cyclic structures (even if the cyclic nature would not hamper parallelization). Other approaches include those based on more traditional dependence analysis (e.g. [Gua88], which assumes that structures do not have cycles) and abstract interpretation techniques (e.g. [Har89], designed for list-like structures commonly used in Scheme programs).

2.2 Our Approach

Based on our past experience of developing alias analysis techniques for list and tree-like structures, and the failure of other techniques to find accurate information for general pointer structures, we believe that a lack of appropriate data structure declarations is the most serious impediment to the further improvement of analysis techniques (and hence the application of optimizing and parallelizing transformations). Thus, we have developed an approach for the abstract description of data structures, called **ADDS**. In this approach the programmer is given a way of describing the properties of his or her data structure in more detail. The ADDS mechanism is a minor addition to existing imperative programming languages (e.g. C) and can be used to describe a wide variety of pointer data structures commonly found in imperative programs. ADDS was designed to: (1) be simple to use for the programmer, (2) handle a wide variety of complex pointer data structures, and (3) provide information that can be effectively utilized by the compiler.

Asking the programmer to specify some properties of his or her data structures should not be considered a radical change in our way of thinking about programming in imperative programming languages. Languages often provide programmers with both one-dimensional and two-dimensional array data types, even though these are both implemented as one-dimensional arrays in memory. A more recent example is Fortran 90 in its treatment of pointers to variables. Variables accessible through pointers must be explicitly declared as

either pointers or targets [MR90]. This simple declaration greatly improves the accuracy of alias analysis in the presence of pointers.

In the pointer data structures domain, programmers already convey quite a bit of implicit information about their data structures. For example, consider the following two recursive type declarations:

```
type BinTree
  { int      data;
    BinTree *left;
    BinTree *right;
  };
```

```
type TwoWayList
  { int        data;
    TwoWayList *next;
    TwoWayList *prev;
  };
```

Even though these type declarations appear identical to the compiler (each declares a record with three fields, one integer and two recursive pointers), the naming conventions imply very different structures to readers of a program. In addition, each structure has some very nice properties which the compiler could exploit. A binary tree naturally subdivides into two disjoint subtrees that can be operated on in parallel. A two-way linked list has the property that a traversal in the forward direction using only the **next** field never visits the same node twice (likewise for traversals using only the **prev** field). This property of never visiting the same node twice enables the parallelization of traversals along the list[1]. The idea of ADDS is simply to make this sort of implicit information explicit to the compiler in order to enable the exploitation of available optimizations and parallelism. Positive side-effects may be increased human understanding of programs, and the compiler's ability to generate run-time checks for the proper use of dynamic data structures.

However, the presence of such a description mechanism alone is not enough. Side-effects in imperative programs often rearrange the components of a data structure, causing a temporary but intentional invalidation of the properties we wish to exploit. Application of optimizing or parallelizing transformations (that rely on these properties) during such a time would be incorrect, and intolerable. Hence some form of data structure validation analysis, beyond alias analysis, is necessary to ensure correctness.

The organization of the remainder of this paper is as follows. In section 3, we summarize our approach to abstract descriptions of pointer data structures (ADDS) and our techniques for providing abstraction validation and alias analysis based on these descriptions. In section 4 we demonstrate the usefulness of our approach by presenting the analysis and transformation of an important class of scientific codes. In particular, we study an N-body simulation program implemented using pointer data structures. Within section 4 we show how the data structure used in such a tree-code based program can be expressed using the ADDS approach, and we investigate what parallelizing transformations can be performed given the improved information available from the ADDS declaration. We also demonstrate good speedups achieved by applying one such transformation on real hardware. Finally, in section 5 we present our conclusions.

[1] Depending on the actual implementation of the list, the *traversal* may still need to be done sequentially (e.g. if the list is built using dynamic allocation). However, the *processing* of each node in the list can be done in parallel. Since processing time generally exceeds traversal time, the effect is a slightly skewed overlap of iterations versus a full overlap. Hence, the important issue is whether nodes can in fact be processed in parallel.

3 ADDS - Abstract Description of Data Structures

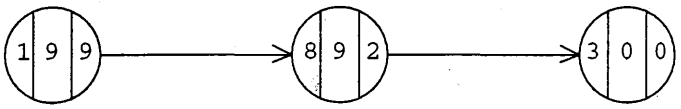
The optimization of codes involving data structures requires knowledge about the properties exhibited by that structure, e.g. shape, size, and method of element access. With arrays, these properties are readily identifiable. Contrast this with user-defined pointer data structures, in which none of these properties are made explicit. Our goal in this section is to first summarize our formalism for expressing the shape of a recursive pointer data structure, a formalism we call **ADDS** (abstract description of data structures). We then summarize a method of analysis—*general path matrix* analysis—which provides abstraction validation and more accurate alias analysis when combined with an ADDS declaration.

3.1 Dimensions and Directions in Pointer Data Structures

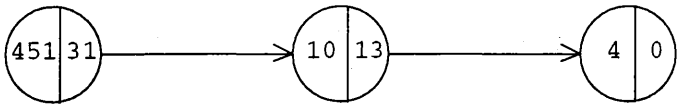
ADDS is a mechanism for describing the *shape* and *traversal properties* of a pointer data structure. This is accomplished by declaring two important properties of the structure—what we call "dimension" and "direction." In this section we shall summarize these notions, and hence ADDS, intuitively through a series of examples taken from scientific applications.

3.1.1 An Introduction to ADDS – Linked Lists

Let us start with one of the most simple and fundamental pointer data structures, the ubiquitous linked-list. Linked-lists have the advantage over arrays in that their size can grow or shrink as needed, and operations like insertion or deletion of elements can be performed more efficiently. One application of linked-lists in scientific programs includes using a one-way linked-list to implement a *bignum* package for integers with "infinite" precision. A bignum can be represented by a list of nodes, where each node in the list contains a fixed number of digits. For example, here is a linked-list representation of the integer 3,298,991 (three digits per node):



Note that the integer is stored in reverse order for ease of manipulation. Another application is the representation of sparse structures like polynomials. For example, the polynomial $451x^{31} + 10x^{13} + 4$ could be stored in a linked-list such that each node contains the coefficient and exponent for x:



In imperative programming languages like C, a linked-list is built out of recursively-defined, dynamically-allocated nodes that have a field for data, and a pointer field that points to the next node. For example, here is a possible type declaration for use in building lists that represent polynomials:

```
type ListNode
 { int       coef, exp;
   ListNode *next;
 };
```

Note that even though we have in mind a one-dimensional, acyclic data structure for our representation of polynomials (and bignums), this type declaration does not express these properties. In fact the data structures shown in Figure 1 could also be built using this same **ListNode** type.

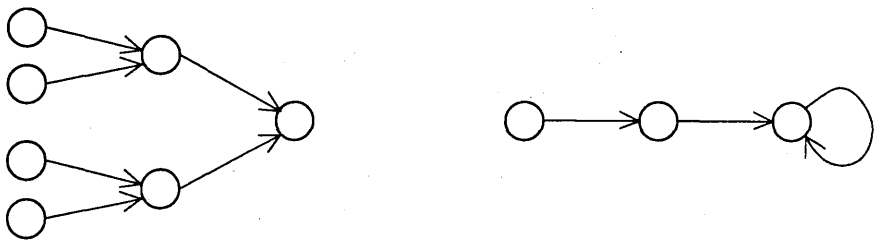


Figure 1: Other possible data structures built using **ListNode**.

Thus, we see that in order for the compiler to exploit the nice properties of the data structure that we intended, we need some mechanism for expressing the appropriate properties. The kind of list that we need for bignums and polynomials is a one-way linked-list as shown in Figure 2.

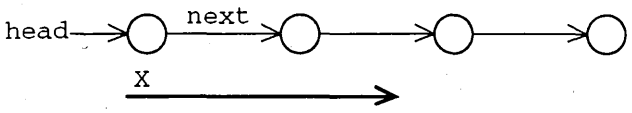


Figure 2: A one-way linked-list.

The one-way linked-list structure obviously has only one "dimension" (that we labeled **X**), and the "origin" of such a list is generally viewed as the head of the list. If we look for the direction of traversal, it is clearly a "forward" traversal away from the origin. Also, such a traversal always moves further away from the origin, implying the structure is free of cycles. Lastly, we note that each node is pointed to by at most one other node; that is, along the **X** dimension, there is a unique path into each node. We capture these properties of a one-way linked-list—namely a single dimension traversed in a unique acyclic direction—in the following ADDS declaration:

```
type OneWayList [X]
 { DataType    data;
   OneWayList *next is uniquely forward along X;
 };
```

This declaration says that a **OneWayList** has only one dimension called **X**. The term **forward** by itself declares an acyclic shape; the notion of "uniquely forward" is used to convey the fact that every node is pointed to by at most one other node along **X** (contrast this with the "tournament" list shown in Figure 1). Hence, this abstract description distinguishes a one-way linked-list from all other data structures that could possibly be built using the same sort of node type.

It should be noted that a programming language and its compilers could directly support pointer data structures such as `OneWayList` via predefined types. However, a quick survey of the literature (or a data structures text such as [Sta80]) would reveal a wide variety of important pointer data structures. Implementations for these structures can differ widely as well. Thus, instead of trying to predefine every possible type of pointer data structure, we chose to develop a technique for allowing the programmer to describe their data structure to the compiler. We believe ADDS is flexible enough to describe the important properties of nearly all pointer data structures.

3.1.2 The Process of Developing an ADDS Declaration

As illustrated by the previous subsection, there is a straight-forward process for developing an ADDS declaration. We summarize this process as follows. Suppose you have a recursive pointer data structure and you wish to describe its shape. Consider the structure in its general form, and select a node as the "origin"—it doesn't matter which node you choose, though some choices make more sense than others (e.g. the root of a tree versus a leaf). Next, think of your structure as having "dimensions," different paths emanating from the origin, with typically one dimension per path. Finally, select a node n other than the origin, and for each recursive pointer field f in n, decide which dimension f traverses and in which "direction." The "forward" direction implies traversing f moves one unit away from the origin, and "backward" implies traversing f moves one unit back towards the origin. A field is limited to traversing one dimension in only one direction[2]. By default, a structure has one dimension D, where it is assumed that all recursive pointer fields traverse D in an "unknown" (i.e. possibly cyclic) direction. As illustrated with the `OneWayList` declaration, the idea of ADDS is to override this default and provide more explicit information.

3.1.3 Developing ADDS Descriptions for Complex Data Structures

The flexibility of the ADDS technique is illustrated by the more exotic recursive pointer data structures. Typically such structures exhibit multiple dimensions, where dimensions are either "independent" (disjoint) or "dependent." For example, an *orthogonal list* [Sta80], used to implement sparse matrices, has two dependent dimensions `X` and `Y` (much like the two-dimensional array it represents). This structure is shown in Figure 3.

For orthogonal lists we note that dimensions `X` and `Y` are dependent since one traversal along `X` and another traversal along `Y` may lead to a common substructure. For example, given the orthogonal list of Figure 3, traversing along `X` from `r4` and along `Y` from `c3` may lead to the same node. One can also think of this property at the node level. Each node n may be accessed by traversing from some node along the `X` dimension, **and** from a different node along the `Y` dimension. This indicates that there is a dependency between the two dimensions. However, even though the dimensions are dependent, orthogonal lists still possess regular properties. For example, traversing forward along `X`, or forward along `Y`, is guaranteed never to visit the same node twice. Further, each row is disjoint, so that parallel traversals of different rows along `X` will never visit the same node (likewise for columns and the Y dimension). These properties are captured by

[2]This restriction can be overcome by the programmer without too much difficulty by using variant records.

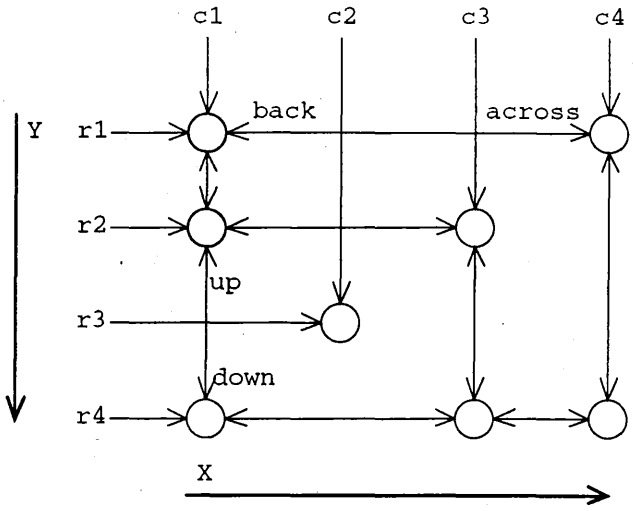


Figure 3: An orthogonal list.

the following ADDS declaration (in particular, note that the fields **across** and **down** are declared as uniquely forward)[3]:

```
type OrthList [X][Y]
 { int       data;
   OrthList *across is uniquely forward along X;
   OrthList *back   is backward along X;
   OrthList *down   is uniquely forward along Y;
   OrthList *up     is backward along Y;
 };
```

An interesting three-dimensional structure that has both dependent and independent dimensions is the *two-dimensional range tree* [Sam90], used to answer queries such as "find all points within the interval x1...x2" or "find all points within the bounding rectangle (x1,y1) and (x2,y2)." As illustrated in Figure 4, a two-dimensional (2-D) range tree is a binary tree of binary trees, where the leaves of each tree are linked together into a two-way linked list.

The important property of a binary tree (and of trees in general) is that for any node n, all subtrees of n are disjoint. Another way of expressing this property of disjointness is that each node in the tree is pointed to by at most one subtree link. In our example of the 2-D range tree, the **down** dimension forms a binary tree in which each node has a different left and right subtree. That is, as we traverse down the tree, at every node we have a unique traversal to the left and a unique traversal to the right.

Now consider the dimensions in the 2-D range tree. There are in fact three ADDS dimensions: **down**, **leaves**, and **sub**. The dimensions **down** and **leaves** are dependent, since each leaf node can be reached by traversing along the **down** dimension and along the **leaves** dimension. However, observe that **sub** is independent of both **down** and **leaves**. That is, any node that can be accessed by a forward traversal along

[3]Also note that unless otherwise stated (via ||), dimensions are assumed to be dependent. Such conservative nature is intentional.

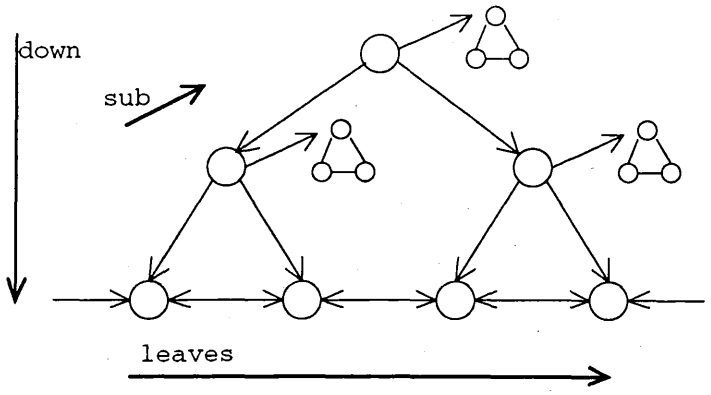


Figure 4: A two-dimensional range tree.

sub, cannot be accessed by a forward traversal along **down** or along **leaves**.

We can express these properties of a 2-D range tree with the following ADDS declaration. Note that by listing the fields **left** and **right** together, we indicate that left and right traversals along **down** are disjoint. Also note how we indicate that the dimension **sub** is independent of **down**, and also independent of **leaves**.

```
type TwoDRangeTree [down][sub][leaves] where sub||down, sub||leaves
 { int            data;
   TwoDRangeTree *left, *right is uniquely forward along down;
   TwoDRangeTree *subtree      is uniquely forward along sub;
   TwoDRangeTree *next         is uniquely forward along leaves;
   TwoDRangeTree *prev         is backward along leaves;
 };
```

For a more formal definition of ADDS and the properties it defines, see [HHN92].

3.2 Speculative Traversability

In all cases, a data structure described using ADDS is required to be *speculatively traversable* [HG92]. This property allows one to traverse past the "end" of a data structure without causing a run-time error. It can be automatically supported by the compiler, and places no additional burden on the programmer (except good programming practices—e.g. in C, they must use the name **NULL** and not an arbitrary integer). This property is analogous to *computing* an array index outside the bounds of an array, but not actually using it. It is often useful when applying optimizing or parallelizing transformations.

3.3 ADDS and General Path Matrix Analysis

The presence of a description mechanism such as ADDS is not enough in itself to enable optimizing and parallelizing transformations in the presence of general (cyclic) pointer data structures. Imperative programs routinely rearrange components of such a structure, and it is during these points in a program that the abstraction (or parts thereof) must be ignored by the compiler. Otherwise transformations may be applied

that are based on invalid assumptions, causing incorrect code generation. Hence some form of pointer analysis is required to validate the abstraction, as well as decide whether a transformation can in fact be applied.

The principal goal of ADDS is to improve the analysis of codes utilizing pointer data structures. As discussed in section 2, existing analysis-only approaches exhibit various limitations when faced with such structures. Our approach is to use the information available in the ADDS declarations to guide the analysis. This synergy between the abstract data structure descriptions and the analysis technique provides a more general and more accurate approach. For example, by using information about the dimensionality and direction of field traversals, the abstraction approximations are freed from estimating needless cycles (such as those formed by the forward and backward directions along the same dimension), and can therefore avoid making needless conservative approximations. Our approach then is a combined one, in which safe analysis techniques are used in conjunction with ADDS declarations. In particular, we are developing an approach to the static analysis of ADDS data structures that is an extension of *path matrix* analysis [HN90, Hen90], called *general path matrix* analysis.

Path matrix analysis was originally designed to automatically discover and exploit the properties of acyclic data structures. General path matrix analysis computes, for each program point, a path matrix PM which estimates the relationship between every pair of live pointer variables. The entry $PM(r, s)$ denotes an explicit path or alias, if any, from the node pointed to by r to the node pointed to by s. The analysis does not attempt to express all possible paths between two nodes, since cyclic data structures would soon overwhelm the matrix. Instead, the paths explicitly traversed by the program are captured in the PM, while the remaining paths and aliases are deduced from the current state of the matrix and the ADDS declarations.

Each new path matrix is generated from the current path matrix. The process is controlled by "pointer rules," which are applied on the basis of the program statement under analysis[4]. For example, there is a rule to handle pointer statements of the form `A->f = B`, which potentially alter a data structure's shape. This is one of the more complex rules, for it must handle DAG (shared) and graph (cyclic) structures. However, the ADDS declaration can be used to simplify this rule, thus making the analysis more efficient. For example, if f is an acyclic field, then the rule developed in [Hen90] can be used, where detection of a cycle denotes a break in the abstraction.

Note that general path matrix analysis fulfills two distinct roles. Firstly, it captures within the PM the current shape of the data structure at each program point. This can be compared with the original ADDS declaration for deciding whether the abstraction is currently valid. Secondly, the PM can be used for alias analysis to determine whether two pointer variables are potential aliases. The former is important for knowing when optimizing and parallelizing transformations are possible, the latter is important for knowing when a possible transformation can be applied.

3.3.1 Abstraction Validation

In order to validate an ADDS declaration, the effect of certain pointer statements on the path matrix must be compared with the original ADDS declaration. In particular, statements of the form `p->f = q` may change

[4] The detailed presentation and discussion of these rules is beyond the scope of this paper.

the shape of the data structure. This in turn may result in a violation of the declared abstraction. However, this is generally not an error in an imperative program, and so is not treated as one. Instead, we note that the abstraction is invalid at this point in the program, and we do not perform any transformations that rely on the validity of the necessary ADDS properties[5].

Though the actual process of validation is beyond the scope of this paper, the idea is as follows. The ADDS declaration is encoded as a series of relationships between the various pointer fields of the node. During analysis, if the path matrix ever denotes a relationship between two fields that is illegal, this part of the abstraction is deemed invalid and an entry is added to the path matrix encoding the violation. Later, if another program statement fixes the relationship between these two fields, the entry is removed and the abstraction is once again considered valid.

A common example of a temporary break in an abstraction is the moving of a subtree from one node to another within a binary tree. Here is a possible code fragment:

```
p1->left = p2->left;
p2->left = NULL;
```

After analysis of the first statement, it is obvious that `p1` and `p2` share a common subtree, even though this violates the disjointness property of a binary tree. However, the violation is immediately corrected, as is usually the case.

3.3.2 Alias Analysis

Alias analysis is best explained via example. Consider the following code fragment. The pointer variable `head` denotes a polynomial represented as a one-way linked-list, where each node has a coefficient, an exponent, and a link to the next node (see section 3.1.1). The code below simply multiplies each coefficient by a constant `c`.

```
p = head;
while p <> NULL
 { p->coef = p->coef * c;
   p = p->next;
 }
```

	$head$	p	p'
$head$	=	=?	=?
p	=?	=	=?
p'	=?	=?	=

If the compiler fails to discover that `next` traverses a list in an acyclic manner, then its analysis of the above code will be overly conservative—the compiler must assume that `next` is cyclic, and hence that `head` and all values of `p` are potential aliases for the same node. The given path matrix illustrates this conservative analysis, where definite aliases are indicated by = and possible aliases are indicated by =?. For example, the path matrix entry $PM[head, p']$, which is =?, indicates that *head* and p' might be aliases (i.e. pointers to the same node). The path matrix contains information about all live pointer variables `head` and `p`, as well as an extra entry for p' which is used to maintain information about the previous iteration. The p' entries are used for detecting aliasing during the loop.

Now suppose the programmer declared his or her linked-list using an ADDS declaration like `OneWayList`, as shown in section 3.1.1. Assuming general path matrix analysis determines that the structure abstraction

[5] Of course, such a violation could in fact be an error. Warning the programmer, or providing a compiler switch to enable these warnings, would be a useful debugging tool.

is valid at the start of this code fragment, the compiler can use the acyclic nature of the **next** field to infer that the statement `p = p->next` always traverses to a different node. For this loop, general path matrix analysis would produce the following path matrices, which denote (from left to right): just before the loop, after one iteration, and after the loop analysis has reached a fixed point.

	head	*p*
head	=	=
p	=	=

	head	*p*	*p'*
head	=	*next*	=
p		=	
p'	=	*next*	=

	head	*p*	*p'*
head	=	$next^+$	$next^+$
p		=	
p'		*next*	=

An entry in a path matrix like $next^+$ indicates a path of one or more **next** links. An empty entry does not necessarily mean there is no path; it does however guarantee that the two pointers are not aliases. Thus the ADDS declaration and the general path matrix analysis have captured the desired property in PM necessary for performing optimizing and parallelizing transformations, namely that **head**, **p**, and p' are never aliases.

4 Parallelization Based on ADDS

In this section we demonstrate the usefulness of ADDS and general path matrix analysis in the parallelization of a class of scientific codes, the so-called *tree-codes* [App85, BH86]. We present the application of general loop parallelization, a loop transformation appropriate for MIMD multiprocessors. Examples of applying other optimizing and parallelizing transformations made possible by using the ADDS approach are given in [HG92] (loop unrolling) and [HHN92] (software pipelining).

4.1 N-Body Simulations using Tree-Codes

N-body simulations are used to study the behavior of N particles as they interact through the force of gravity or other forces [App85]. For example, these particles can be bodies in space [BH86] or water molecules [LC86]. The simulation typically runs iteratively, applying the following algorithm at each time step:

```
for each particle p /* L1 */
  compute the force on p due to other particles;
for each particle p /* L2 */
  compute new velocity and position of p;
```

We shall refer to these as loop #1 (L1) and loop #2 (L2). The obvious implementation uses an array of particles, where L1 considers the interactions between all particles. Once L1 is complete, L2 is a simple iterative loop through the array to update each particle. L1 dominates the computation, resulting in a $O(N^2)$ sequential algorithm.

A more elegant and efficient approach was first presented by Appel [App85], and made more robust by Barnes and Hut [BH86]. It is based on the observation that a collection of particles can be approximated as a point mass, and thus the computation of a single interaction can often replace that of computing all interactions with the constituent particles. A tree is used to represent the system, where the leaves denote the particles and the interior nodes denote the point masses. The result is a $O(NlogN)$ sequential algorithm[6],

[6] Algorithms with better time complexities exist, but with various limitations. See [App85] and [Mak90].

commonly referred to as the *Barnes-Hut* algorithm [BH86].

The original Barnes-Hut algorithm can be viewed as follows. The data structure is an *octree* (each node has at most eight subtrees, denoting quadrants in space around the node), where the leaves (original particles) form a one-way linked-list. We give an illustration of a typical octree in Figure 5.

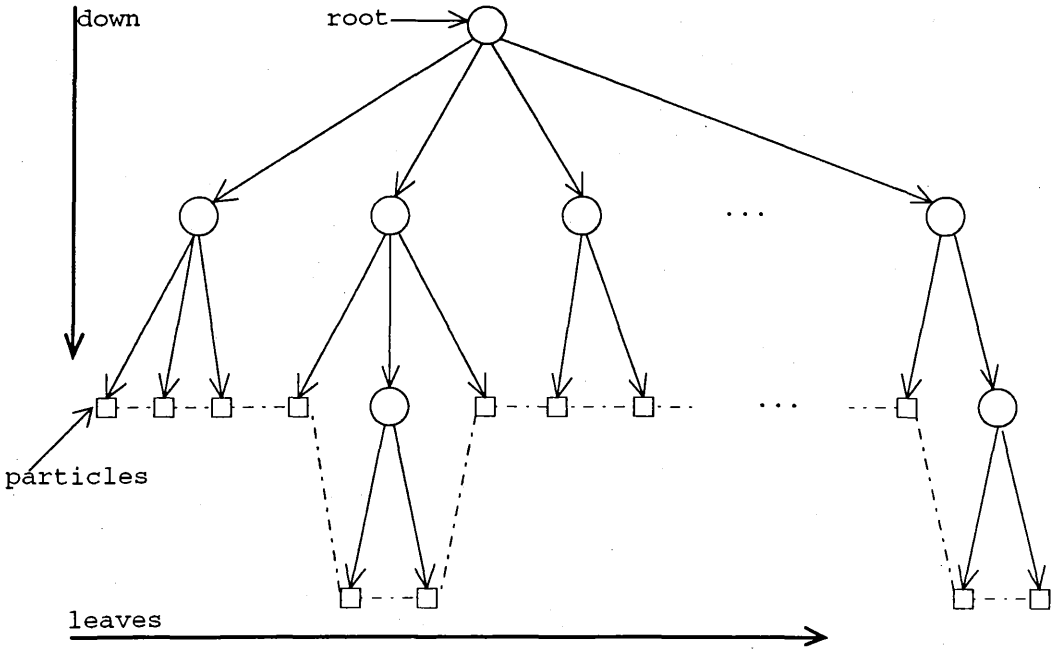


Figure 5: An octree.

Since L2 computes new particle positions and velocities, note that the tree must be rebuilt at the start of the next time step. Hence at each time step, the following algorithm is applied:

```
root = build_tree(particles);
p = particles;
while p <> NULL  /* BHL1 */
 { p->force = compute_force(p, root);
   p = p->next;
 }
p = particles;
while p <> NULL  /* BHL2 */
 { compute_new_vel_pos(p);
   p = p->next;
 }
```

The function **compute_force** recursively descends the tree, finding nodes to include in the force calculation. The procedure **compute_new_vel_pos** updates the velocity and position vectors given the new force upon the particle.

```
function compute_force (p, node)
 { if p and node are WELL-SEPARATED
     then
       return force computed using node;
     else
       return the sum of calling compute_force on subtrees;
 }

procedure compute_new_vel_pos (p)
 { compute change in p's velocity and position;
   update p's velocity vector;
   update p's position vector;
 }
```

Note that once a node is included in the force computation, its subtrees are ignored. More importantly, note that each iteration of BHL1 is independent, except for the loop-carried dependency of the list traversal (the same is true for BHL2). Further, the subtree traversals within `compute_force` are likewise independent from one another. Hence the Barnes-Hut algorithm possesses large amounts of parallelism.

4.2 Parallelization of Tree-Codes

Though the original Barnes-Hut algorithm is elegant, efficient, and easy to understand as a tree-structured problem, in terms of parallelization it suffered from two significant disadvantages: (1) use of pointer data structures instead of arrays, and (2) use of recursion over iteration. We shall focus on the first disadvantage here; recursion can be handled using the techniques discussed in [HN90, Hen90].

As discussed in sections 1 and 2, optimizing compilers are overly conservative in the presence of pointers. For example, consider BHL1. The call to `compute_force` implements a recursive summation of forces, stored in the corresponding leaf node. Conventional parallelizing compilers will be unable to determine that `p = p->next` always points `p` to a different node. As a result, the compiler must assume that `p` and `p->next` are potential aliases. Likewise for BHL2. This aliasing in turn creates false dependencies between loop iterations, in fact between all iterations, complicating and most often preventing the application of semantic-preserving transformations on these loops.

As a result, users of the Barnes-Hut algorithm have changed the implementation in a number of ways in order to improve performance on vector and multiprocessor architectures. In essence, arrays are used whenever possible (e.g. see [Her90, Bar90]), and recursion is replaced by iteration (e.g. using arrays [Mak90]). As an aside, the *Water* benchmark from the SPLASH suite [SWG91] is a similar N-body simulator of water molecules. It is based however on a $O(N^2)$ algorithm using arrays and iteration, most likely for ease of parallelization.

Obviously, reworking such a program by hand to increase parallelism requires a non-cursory understanding of both the code and its data structures. Such changes also hinder readability, debugging, and portability. The preferred approach of course would leave the original implementation relatively intact and automatically transform the code to take advantage of the available parallelism. This requires a more accurate analysis of the code, which in turn requires an understanding of the octree and its properties. ADDS can be used to provide the necessary understanding, and general path matrix analysis the necessary analysis. As we

shall see, use of ADDS requires only minimal change to the data structure declarations, and arguably may increase readability and enhance debugging.

4.3 Parallelization of the Barnes-Hut Algorithm

To demonstrate the usefulness of our proposed approach, we will consider the original Barnes-Hut algorithm as described earlier. Our goal in this paper is to describe the parallelization of the two loops, BHL1 and BHL2.

4.3.1 Declaring the Octree

The first step, and the only change required to the program, is the declaration of the octree data structure using ADDS. This is supplied by the programmer as follows:

```
type Octree [down][leaves]
 {  LocalData  d;
    boolean    node_type;
    Octree    *subtrees[8] is uniquely forward along down;
    Octree    *next        is uniquely forward along leaves;
 };
```

Referring back to Figure 5, we can see how this ADDS declaration was developed. There are clearly two different dimensions for the structure; the **down** dimension, used to traverse down through the tree, and the **leaves** dimension, used to traverse across the leaves. The natural origins are the **root** node for the **down** dimension, and the leftmost leaf node for the **leaves** dimension. It is also clear that with respect to these origins, the **subtrees** links traverse **forward** along the **down** dimension, and the **next** link traverses forward along the **leaves** dimension. The forward traversals along the **down** dimension are unique because each node may be pointed to by at most one **subtrees** link. Similarly, the forward traversal along **leaves** is unique because each leaf node is pointed to by at most one **next** link. Finally, we can determine that the two dimensions are dependent (the default) because it is possible to reach a node along both the **leaves** dimension and the **down** dimension. In summary, the two important properties implied by this ADDS declaration are that given any node: (1) each of its subtrees are disjoint with respect to traversals along the **down** dimension, and (2) traversals along the **leaves** dimension never visit the same node twice.

4.3.2 Analysis of the Program

As discussed in section 3.3, two different analyses are required before any parallelizing transformation may be applied: abstraction validation and alias analysis. We shall discuss the latter first, and consider only BHL1; analysis of BHL2 is identical.

Assume the ADDS declaration is valid when BHL1 is reached. The goal of general path matrix analysis is to discover the absence of aliasing within each loop, i.e. show that no two iterations write to the same node n. This is easily done by (1) observing that `compute_force` writes only to the node denoted by `p`, (2) observing that BHL1 does not rearrange the data structure, and (3) given that the `Octree` declaration

is valid, knowing that `p = p->next` always refers to a different node. The result is a path matrix for BHL1 similar to the one shown earlier in section 3.3.2:

	root	*particles*	*p*	*p'*
root	=	=?	=?	=?
particles	=?	=	$next^+$	$next^+$
p	=?		=	
p'	=?		*next*	=

This states that for all iterations of BHL1, **particles**, p, and p' are never aliases. It also states that **root** is a possible alias with all other pointer variables; this will not present a problem however since analysis of **compute_force** would show that the data accessed via **root** (and all nodes derived from **root**) are used in a read-only manner.

Confirming that the **Octree** declaration is in fact valid when BHL1 is reached requires analysis of the **build_tree** function (along with analysis of the code that builds **particles**—the process of validation will be similar, so we assume it has already been performed). This function builds the tree in a bottom-up fashion, by first expanding the system's "box" in space to encompass the new particle, and then inserting the particle within this box such that it is the only particle within its "quadrant."

```
function build_tree (particles)
 { p = particles;
   root = NULL;
   while p <> NULL
    { root = expand_box(p, root);
      insert_particle(p, root);
      p = p->next;
    }
   return root;
 }
```

The function **expand_box** extends the tree upward, adding nodes until the tree represents a space large enough to include p. Then **insert_particle** goes down the tree, looking for p's quadrant in this space; if the quadrant is occupied by another particle, the quadrant is subdivided (i.e. new subtrees are added) until the two particles fall in different quadrants.

In order to validate the **Octree** abstraction, general path matrix analysis must compare the path matrix at each program point within **build_tree**, **expand_box**, and **insert_particle** to the ADDS declaration. The actual step-by-step analysis, with presentation of the path matrices, is not the main topic of this paper, and so we provide only a high-level summary of the important points of such an analysis.

First, **expand_box** extends the tree by allocating a node(s) and then making the current tree a subtree. The path matrix will show that root is only pointed to by the variable **root**, and so allocating a new node, storing root as a subtree, and then making this new node the root will leave (1) all subtrees disjoint, and (2) **root** still the only pointer to the root of the tree.

Next, **insert_particle** is a loop that descends the tree looking for p's quadrant. If the quadrant is empty (NULL), then p is stored in this field; obviously the subtrees remain disjoint in this case. If however p's quadrant is already occupied, then subtrees are created until it's empty. In this case there may be a

period in which the abstraction is broken, for a new node is allocated as the subtree, and then p and its competitor are stored as children of this new subtree (assuming each has a quadrant to itself, otherwise the process is repeated). But recall a pointer to the competitor still remains in the original tree, since this is what triggered the creation of the new subtree. Hence a subtree is being shared. However, in the final step the new subtree replaces the competitor in the original tree, and so the abstraction is again valid. This sharing can be handled by a temporary addition to the path matrix, which is later removed (similar to the way DAGS are handled in [HN90, Hen90]).

Finally, assuming the abstraction holds when **build_tree** is entered, analysis will confirm that each iteration of the loop processes a different node (the path matrix will look similar to those shown in section 3.3.2). Hence, all subtrees remain distinct. And since the **next** field is never updated in any of these subroutines, the analysis can conclude that the **Octree** declaration will be valid upon return from **build_tree**.

4.3.3 Transformation of BHL1 and BHL2

Based on the ADDS declaration and the previous analysis, it can be determined that each iteration of BHL1 (and BHL2) is independent, save the loop-carried dependency arising from `p = p->next`. As a result, BHL1 can be parallelized by strip-mining by the number of processors, and then running the inner loop in parallel[7]. In other words, each iteration of BHL1 now processes n particles in parallel, where n is the number of available processing elements. PE 0 will process the particle denoted by p, PE 1 will process `p->next`, PE 2 will process `p->next->next`, and so on. After one parallel iteration, p is skipped ahead n particles, and the process is repeated:

```
p = particles;
while p <> NULL
 { for i = 0 to PEs-1 in parallel
     _BHL1_iteration(i, p, root);
   for i = 0 to PEs-1  /* FOR1 */
     p = p->next;
 }
```

```
procedure _BHL1_iteration(i, p, root)
 { for k = 1 to i  /* FOR2 */
     p = p->next;
   if p <> NULL
     then p->force =
                compute_force(p, root);
 }
```

The same process can be used to parallelize BHL2. The reader may be wondering why the loops **FOR1** and **FOR2** do not cause run-time errors, since it appears that these loops may possibly traverse off the end of the **particles** list. As discussed in section 3.2, ADDS data structures are *speculatively traversable*, which allows traversal off the end of a structure (much like computing an array index outside the bounds of an array, without *using* it). This property is used in the transformation to avoid excess checks for NULL pointers.

4.4 Results

In order to measure the benefit of these transformations, we ran the newly transformed Barnes-Hut algorithm on a Sequent multiprocessor. The original sequential version was timed, along with the transformed parallel version on 4 and 7 processors. All times represent seconds, with simulation runs of 80 time steps.

[7] An obvious transformation, given knowledge about the data structure, which is exactly our point.

TIMES	$N = 128$	$N = 512$	$N = 1024$
seq	188	1496	3768
par(4)	75	548	1343
par(7)	57	369	873

SPEEDUP	$N = 128$	$N = 512$	$N = 1024$
seq	1	1	1
par(4)	2.5	2.7	2.8
par(7)	3.3	4.1	4.3

Though the speedups are not linear, they are quite good given that (1) simple static scheduling is being used, (2) the parallelism inherent in the independent subtree computations (within `compute_force`) is not yet being exploited, (3) synchronization on a Sequent is rather slow, and (4) no attempt is made to optimize the granularity of iterations. Note that runs with N=1024 are not considered large in typical applications.

5 Conclusions

As we have demonstrated with numerous examples in this paper, there are interesting and efficient scientific applications which make use of recursively-defined pointer data structures. With the increasing use of C and Fortran 90 for scientific applications, the desire to use such data structures will no doubt increase. Hence the need for analysis and parallelization of such codes will increase as well.

Although difficult to analyze with traditional methods, many pointer data structures exhibit important properties which can be exploited for optimization and parallelization purposes. Although these properties are often known to the programmer (in imperative programming languages they are often conveyed implicitly via appropriate identifiers or comments), the properties are not available to the compiler. This lack of information hinders the development of accurate alias analysis and thus restricts the transformations that can be applied to codes using pointer data structures. In this paper we have illustrated how our abstract description technique can be used to accurately describe a wide variety of scientific pointer data structures, including one-way linked-lists, orthogonal lists, range trees, and octrees. These examples demonstrate that the development of ADDS declarations is quite intuitive, and does not place an excessive burden on the programmer.

Combined with an extended form of path matrix analysis, called *general path matrix* analysis, our approach enables accurate alias analysis and hence the application of numerous optimizing and parallelizing transformations. In this paper we concentrated on developing the parallelization of a class of scientific codes known as tree-codes. We provided a detailed development of how to apply our technique to an N-body simulation, and we provided concrete experimental results to demonstrate that such parallelization leads to good speedups.

ADDS is a small addition to an imperative programming language, but as illustrated in this paper, it can lead to analysis and parallelization methods that are otherwise not possible for programs that use pointer data structures.

References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[AN88] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, June 1988.

[App85] Andrew W. Appel. An Efficient Program for Many-Body Simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[ASU87] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[Bar90] Joshua E. Barnes. A Modified Tree Code: Don't Laugh; It Runs. *Journal of Computational Physics*, 87:161–170, 1990.

[BH86] Josh Barnes and Piet Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, 4 December 1986. The code can be obtained from Prof. Barnes at the University of Hawaii.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[DH79] J.J. Dongarra and A.R. Hinds. Unrolling loops in FORTRAN. *Software-Practice and Experience*, 9:219–226, 1979.

[Gua88] Vincent A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. Technical Report CSRD Rpt. No. 860, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, February 1989.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, April 1990. TR 90-1114.

[Her90] Lars Hernquist. Vectorization of Tree Traversals. *Journal of Computational Physics*, 87:137–147, 1990.

[HG92] Laurie J. Hendren and Guang R. Gao. Designing Programming Languages for Analyzability: A Fresh Look at Pointer Data Structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages (to appear, also available as ACAPS Technical Memo 28, McGill University)*, April 1992.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, January 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations: Volume I.* Wiley, 1978.

[Lam88] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[LC86] G.C. Lie and E. Clementi. Molecular-Dynamics Simulation of Liquid Water with an *ab initio* Flexible Water-Water Interaction Potential. *Physical Review*, A33:2679 ff., 1986.

[LH88a] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LH88b] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[Mak90] Junichiro Makino. Vectorization of a Treecode. *Journal of Computational Physics*, 87:148–160, 1990.

[MR90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, 1990.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[RG82] B. R. Rau and C. D. Glaeser. Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support. In *Proceedings of the 9th Symposium on Computer Architecture*, April 1982.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[Sta80] Thomas A. Standish. *Data Structure Techniques.* Addison-Wesley, 1980.

[SWG91] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991. Available via FTP from mojave.stanford.edu.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.